Net Elements and Annotations for Computer Programming: Computations and Interactions in PDF

John Frederick Chionglo

Aespen. Markham, Ontario Canada L6B 1B7
john.chionglo@aespen.ca

Abstract

A visual programming language may help a computer programmer develop computer programs [9]. Diagrams based on Petri's net elements have been used to model systems from a wide range of disciplines for control, simulation, analysis and communication [5, 12, 15, 17]. Diagrams based on Holt's net elements, an extension to Petri's net elements, may be used to organize parts of a computer program. The system of computer program organization or annotation system considered here includes Holt's net elements, annotation classes, a system for creating names for identifiers, and a form view of an annotations database. With a Petri Net diagram and this annotation system, it should be possible for a computer programmer to systematically develop computer programs. The computer programs considered here are JavaScript™ programs in PDF documents; they interact with some of the objects of the Acrobat™/JavaScript API to realize token games and other form-based applications.

Introduction

Net Coding Workflow (Figure 1). Annotations of net elements may be used to organize logic computations such as parts of a computer program [19]. Examples of logic annotations include tests conditions for inputs and transitions; firing rules for transitions; and weight computations for input or output arcs [7, 8, 21].

The logic annotations of net elements include computations of net element properties or call on logic annotations of corresponding or related net elements. They may also call on or receive calls from view annotations. The view annotations of net elements interface with the displays and events (user and system) of a host environment.

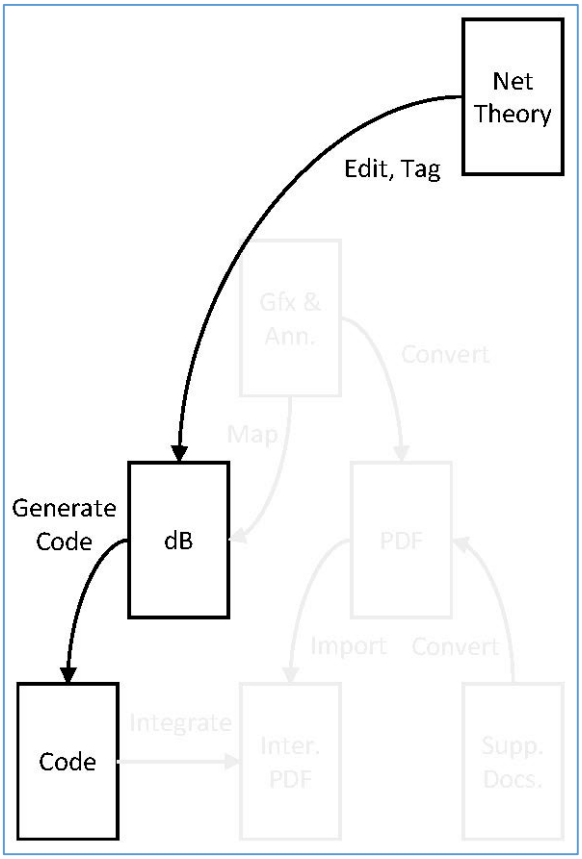

Figure 1 Net Coding Workflow

It should be possible to maintain a database (dB) of the net elements and their annotations directly or via a form. The assembly of logic annotations of net elements and of view annotations from the database is part of an application's computer program (Code).

Net Programming Workflow (Figure 2). A visual programming language may help a computer programmer develop computer programs [9]. Instead of working directly with the database or a form, a computer programmer may use a graphics-based editor to create, edit and annotate net elements. Every graphics object in the editor stands for a net element or an annotation. The objects should map to the database.

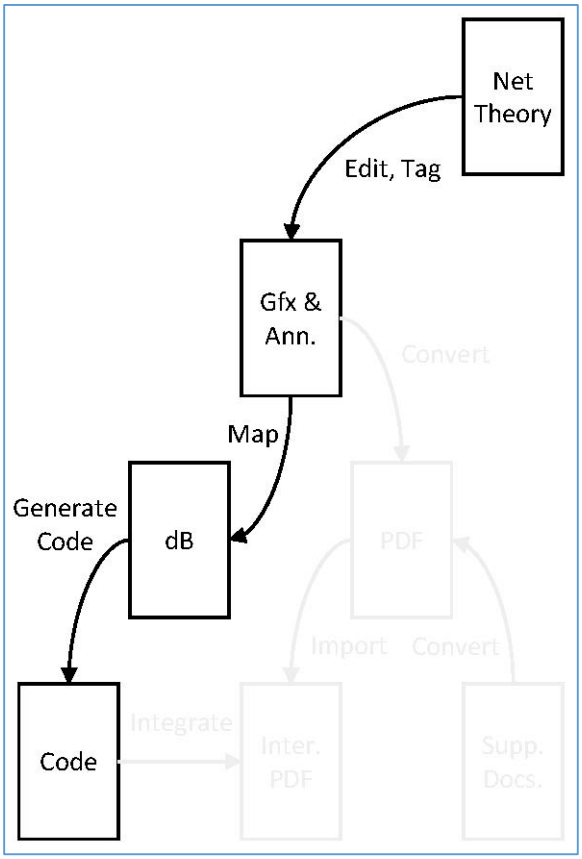

Figure 2 Net Programming Workflow

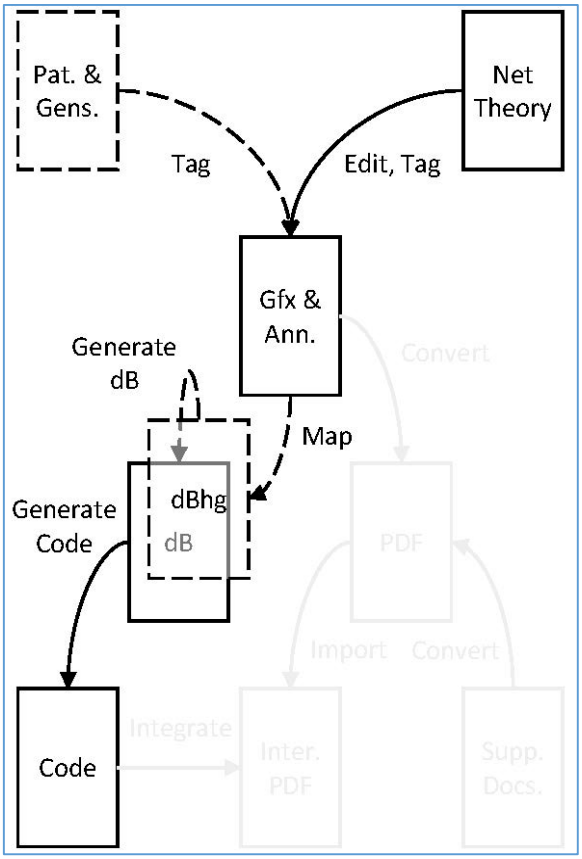

Figure 3 Net Programming Workflow with Generators and Patterns

Net Programming Workflow with Generators and Patterns (Figure 3). For descriptions with too many elements or patterns of logic or repeated components, it would be challenging to work directly with the database, to fill the forms, or to use a graphics editor [16, 19]. To mitigate the challenge, a graphics object may represent more than one net element of the same type [16]; it may also represent more than one net element of different types [20]. An element generator, a special type of annotation, may be used to map a graphics object to more than one net element. An element pattern, a set of commonly used element generators, may be used to facilitate the application of several element generators. A logic generator, another special type of annotation, may be used with an element generator to create logic annotations. A logic pattern may be used to facilitate the application of several logic generators.

Net Integration Workflow (Figure 4). The computer programs considered here are JavaScript™ programs in PDF documents; they interact with some of the objects of the Acrobat™/JavaScript API (such as Field, Doc and app) to realize token games and other form-based applications [1, 2, 6].

The computer program (Code) is integrated into the master PDF document (Inter. PDF). Some of the objects in the graphics editor (Gfx & Ann.) and supporting documents (Supp. Docs.) are converted to PDF documents, and imported into the master PDF document. Utility programs (Utils.) integrate codes. Code integration includes the importation of computer codes (e.g. the importation of JavaScript programs as document-level scripts) and supporting PDF documents; and the one-time creation of objects (e.g. the addition of field objects).

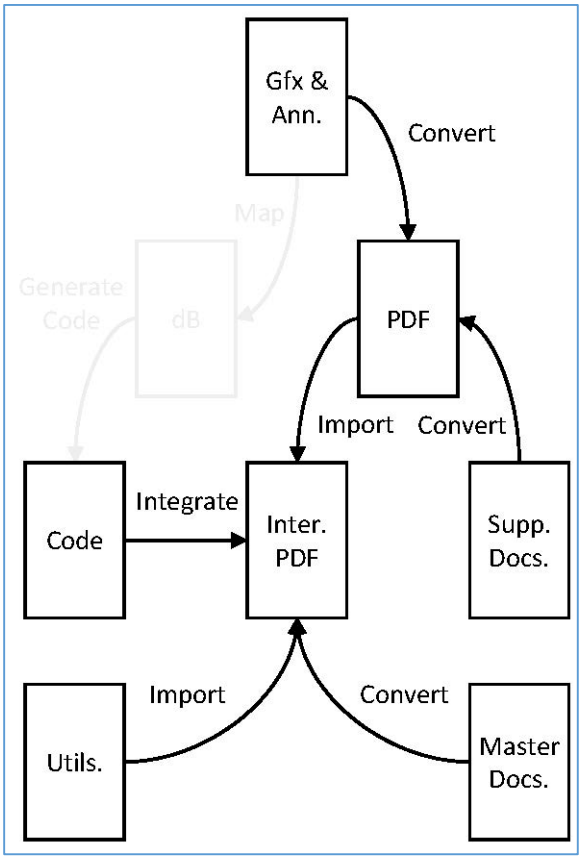

Figure 4 Net Integration Workflow

Net Programming and Integration Workflows (Figure 5). The system of computer program organization considered here supports the workflows for net programming using JavaScript, and integration with PDF using the Acrobat/JavaScript API. It may be used as a basis for visual programming languages designed to realize token games and other form-based applications in PDF.

Net Viewing and Interactions (Figure 6**).** For token games and other form-based applications in PDF, the viewing experience of a user is an interactive process. The interaction includes activities that a user may not or cannot perceive. Examples of activities are *initialization* (*i*) activities triggered by the opening of a PDF document, *process* (*p*) activities triggered by user or system events, *computation* (*c*) activities for updating properties of net elements, and *update* (*u*) activities for notifying the host environment of related changes [1, 6].

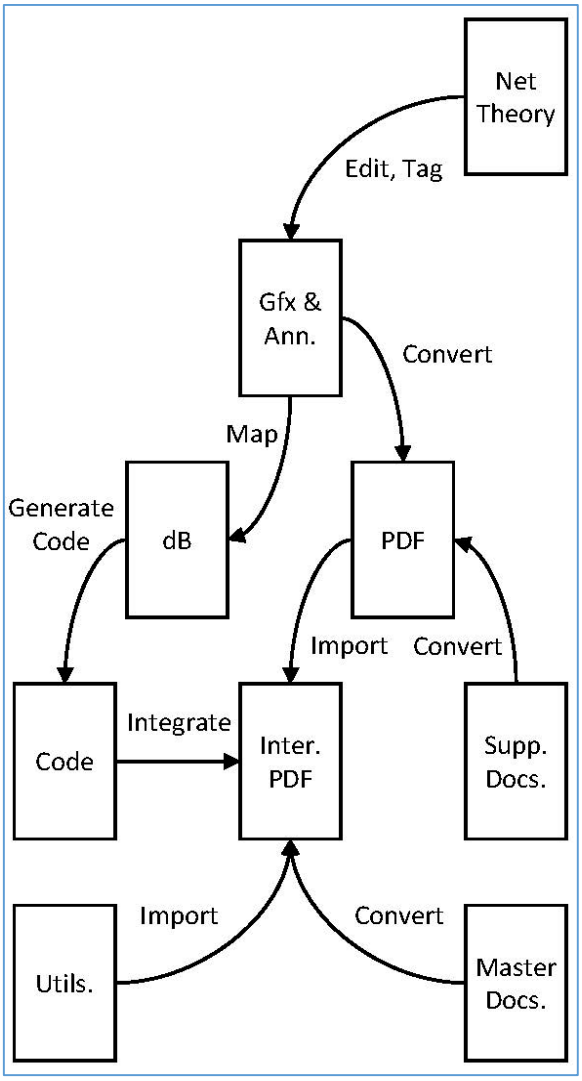

Figure 5 Net Programming and Integration Workflows

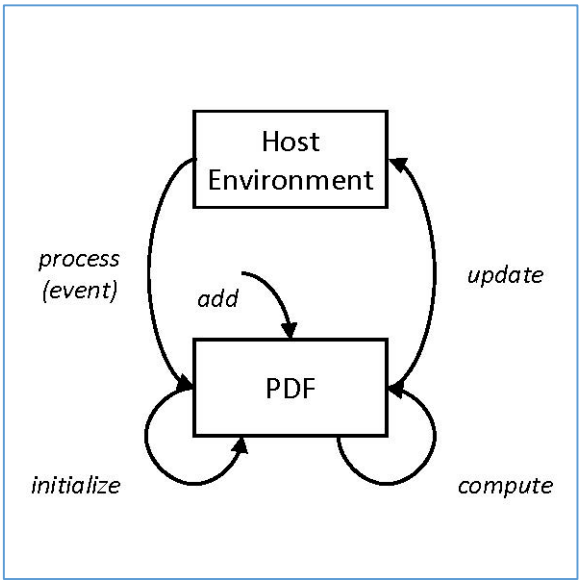

Figure 6 Net Viewing and Interactions

Net Theory

Petri's Net Theory. The elements of Petri's net are place, transition, input and output – an input connects a place to a transition and an output connects a transition to a place [10, 20]. Every net element may have an inscription, a tag or an annotation [19]. Net elements and net element inscriptions may be used to describe a wide range of systems at various levels of abstractions and for many purposes – such as control, simulation, analysis and communication [19]. For example, they may be used to describe system components in prose, to present system parts using graphics, and to specify system modules in terms of mathematical expressions or computer codes [19].

Net Elements

Holt's net elements. The net elements considered here are place (P), transition (T), input (In), output (Ou), runners (R) and net (N). In other words, Holt's net elements are Petri's net elements plus the runner and the net. The runner is an object that is responsible for a set of transitions; it may be associated with a computer processor. Every net element has a unique identifier (ID). Every place (P) as a place identifier (pID). Every transition (T) has a transition identifier (tID). Every input (In) has an input identifier (eID). An input connects a place to a transition – thus it has a related input place (pID) and transition (tID). Every output (Ou) has an output identifier (eID). An output connects a transition to a place – thus it has a related output place (pID) and transition (tID). Every runner (R) has a runner identifier (eID). And the net (N) has a net identifier (n)

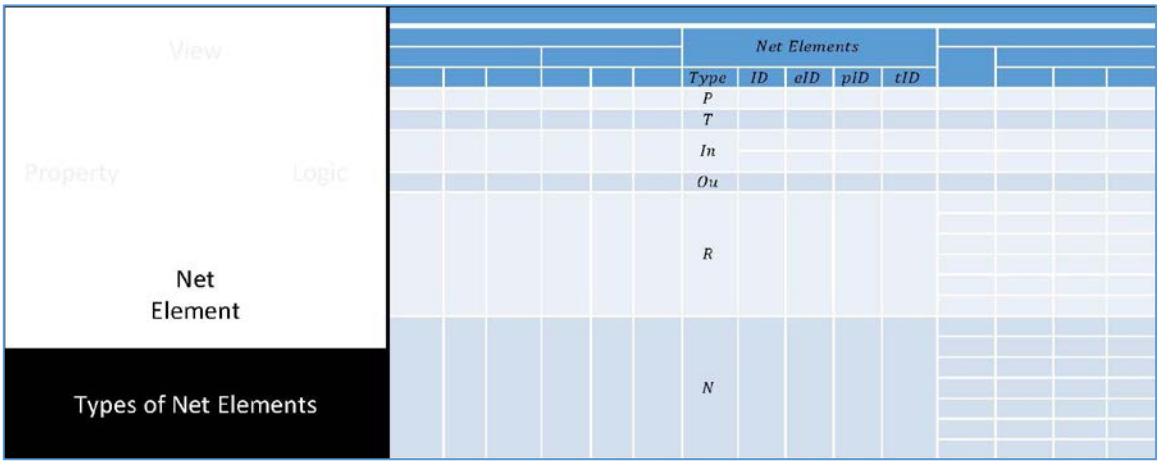

Figure 7 Net Elements

Annotation System

This annotation system can be used as a guide to create a wide variety of token games and other form-based applications. It is based on the author's experiences (from 2011 – 2014) in creating token games for Inhibitor Nets or nets with flexible arcs using JavaScript in PDF; in using several objects (such as the Field, Doc and app objects) of the Acrobat/JavaScript API; and in using Word, PowerPoint and Acrobat Pro [1, 2, 4, 6, 13, 14]. With this annotation system a computer programmer or software developer or software engineer should be able to methodically specify custom annotations for updating properties of net elements, and visualizing and interacting with net elements and net element properties. It is a system for organizing net element annotations.

Classes of Annotations

There are six classes of annotations. First, property annotations of net elements. Second, logic annotations of net elements. Third, view annotations of net elements. Fourth, logic annotations of view annotations of net elements. Fifth, view annotations of property annotations of net elements. And sixth, logic annotations of view annotations of net elements and of property annotations of net elements. Every

element may have any number of annotations. However it is unnecessary for an element to have every available annotation. Thus annotations should be selected for each net element to satisfy a particular token game or form-based application. The following selection is a set of annotations suitable for creating token games.

Property Annotations of Net Elements. A place has a mark (m). A transition has a status (s^t). Every input has a weight (w^-) and a status (s^-). Every output has a weight (w^+) and may have a status (s^+). The values of properties m, w^-, and w^+ may come from any domain such as the sets of Natural Numbers, Real Numbers, and *n*-bit Floating Point Numbers; or they may come from structured information such as the ones used in Predicate/Transition Nets [7], Coloured Petri Nets [8] and Algebraic Nets [21]. A status can be enabled or not enabled. If the status of a net element is enabled, the net element can fire. If the status of a net element is not enabled, the net element cannot fire. A runner may have a status (s^r) and it is responsible for one or more transitions (t^r). Also it has one or more run logic (rn), one or more start logics (st), one or more stop logics (sp), one or more running frequencies (ms), and one or more interval or timeout objects (ai). A run logic is a sequence of steps for firing one or more transitions and performing necessary updates. A start logic is a sequence of steps for starting an interval or timeout object. A stop logic is a sequence of steps for stopping an interval or timeout object. A running frequency is an amount of time in milliseconds between the call of a run logic and the next call of the same run logic. The Net may have a net (n) property. It also has a document (dc) and the app (ap) properties [2].

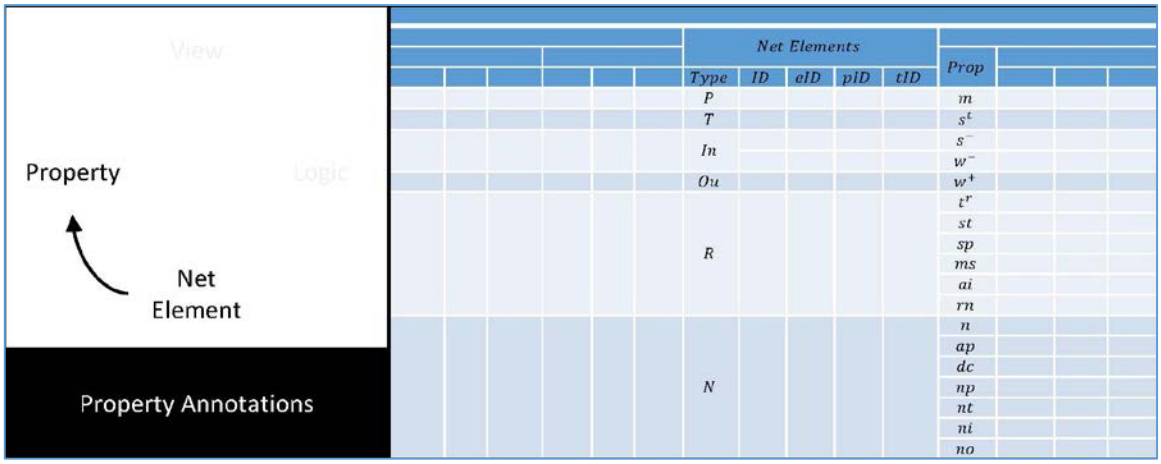

Figure 8 Property Annotations

Logic Annotations of Net Elements. The *Is Enabled* (iE) logic annotation updates the status of a net element or calls on the *Is Enabled* logic annotation of related net elements. References to the *Is Enabled* annotation for transition, input and output are iE^t, iE^- and iE^+ respectively. The *Fire* (F) logic annotation updates the mark of a related place or calls on the *Fire* logic annotation of related inputs and outputs. References to the *Fire* logic annotation for transition, input and output are F^t, F^- and F^+ respectively. The *Get Weight* (gW) logic annotation updates the weight of an input or an output [21]. It may also call on the *Get Weight* logic annotation of related inputs and outputs. References to the *Get Weight* logic annotation for transition, input and output are gW^t, gW^- and gW^+ respectively. The *Start* (st) logic annotation begins the interval or timeout object of a run in a runner [2]. The *Stop* (sp) logic annotation ends the interval or timeout object of a run in a runner [2]. The *Run* (rn) logic annotation sends a fire message to one or more enabled transitions of the runner and sends related update messages.

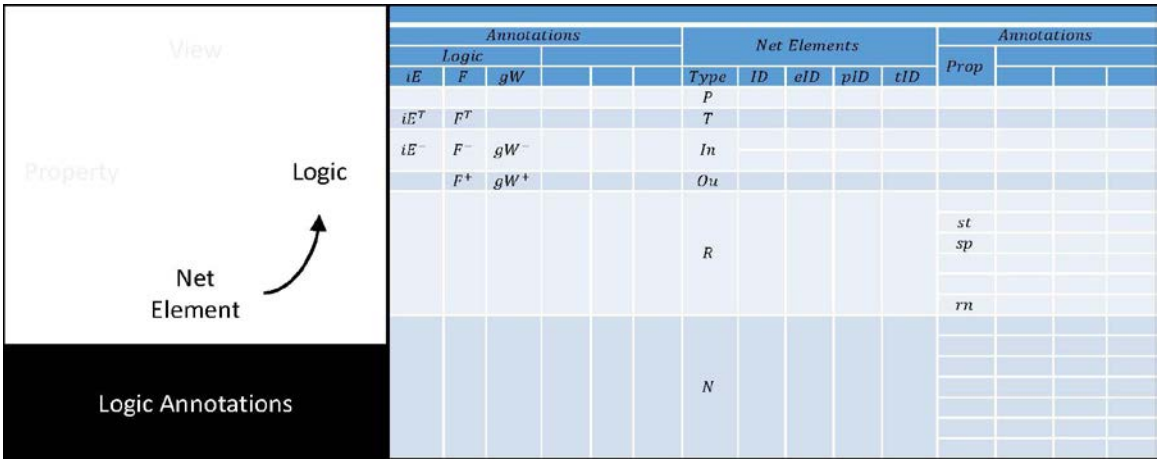

Figure 9 Logic Annotations

View Annotations of Net Elements. The view annotations of net elements interface with the display or events of a host system. Each of the net elements may have an *Event* (Ev), *Label* (L) or *Graphics* (Gx) annotation. The Event annotation is used to capture and process user events. The Label annotation is a visual text that should uniquely identify a net element. The *Graphics* annotation is a visual representation of a net element.

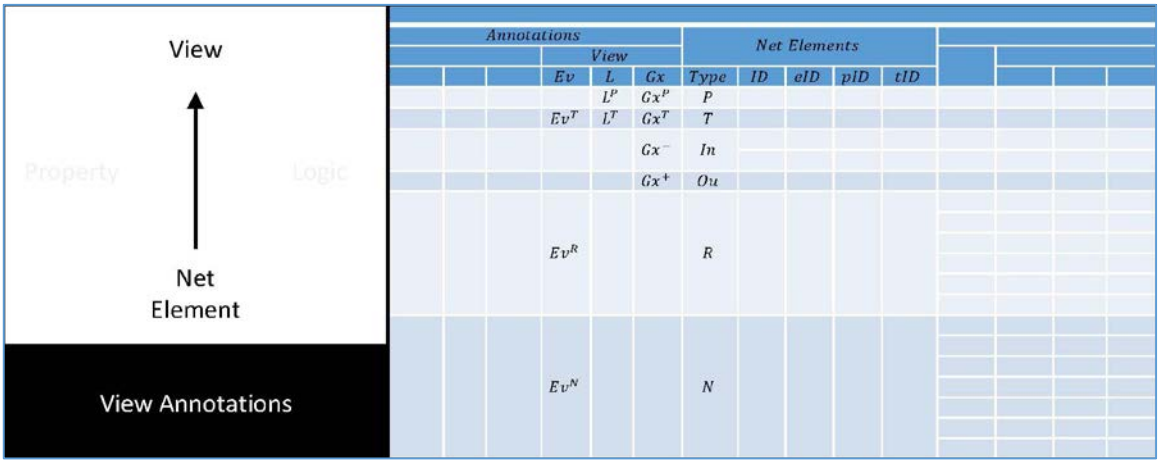

Figure 10 View Annotations

Logic Annotation of View Annotations of Net Elements.

The *Event* (Ev) annotation is the logic annotation of view annotations of net elements. The logic annotation is the computation that must be performed when the related user event or system event occurs. The event annotation of a transition or a net responds to one or more user events. The event annotation of a runner responds to one or more system events. A user event is an occurrence based on end user actions – such as a mouse event. A system event is an occurrence based on a computer system action – such as a timing event. The event annotation is based on the Field object (type: button) [1, 2, 3].

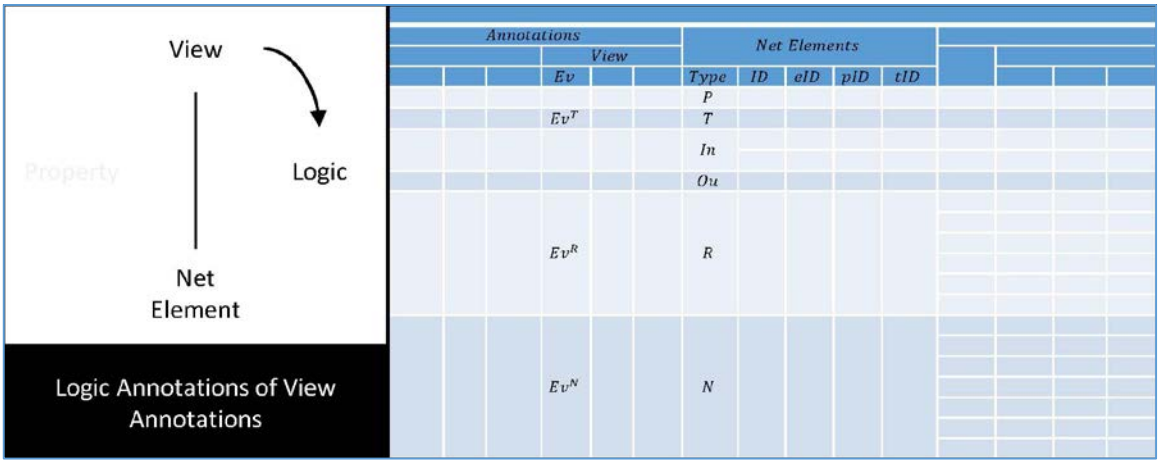

Figure 11 Logic Annotations of View Annotations

View Annotations of Property Annotations. Property annotations may have zero or more view annotations: *Text* (Tx), *Icon* (Ic) and *Display* (D). The view annotation of a property should interface with an external system or host environment. A text annotation is a textual visualization of a property annotation. A display annotation is a graphic visualization of a property annotation. An icon annotation is a source of images to the corresponding display annotation. These annotations are based on the Field object (type: text, button) [1, 2, 3].

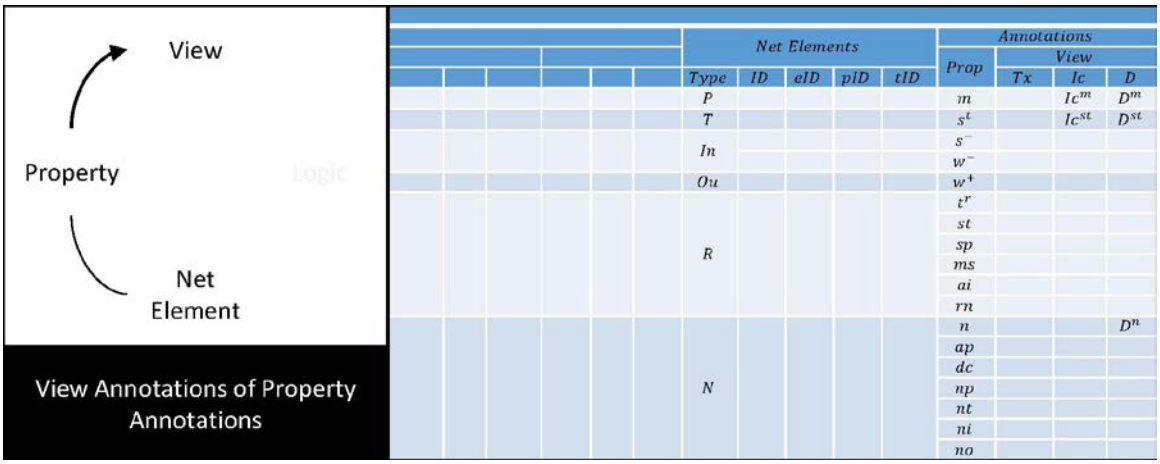

Figure 12 View Annotations of Property Annotations

Interactions of View Annotations. Every view annotation (Ev, Tx, Ic, D) is an interface to the host environment. It is based on the Field object (type: text, button) [1, 2, 3]. It may be added to a PDF document (a), initialized when its PDF document is opened (i), and may update its host when needed (u). System and user events may also be processed (pEv).

		Annotation						
		Logic			View			
		iE	F	gW	Ev	Tx	Ic	D
Interaction	a				■	■	■	■
	i				■	■	■	■
	p				■			
	u				■	■		■
	c	■	■	■				

Table 1 Annotation versus Interaction

For the token games considered here, every place mark has a display view (D^m). Each display view must have an initialize (iD^m) and update (uD^m) logic; it has a source icon (Ic^m) that must have an initialize (iIc^m) logic. Every transition status has a display view (D^{st}). Each display view must have an initialize (iD^{st}) and update (uD^{st}) logic; it has a source icon (Ic^{st}) that must have an initialize (iIc^{st}) logic. The event annotation of a transition must have initialize (iEv^{st}), process (pEv^{st}) and update (uEv^{st}) logic; it

has a source icon (Ic^{st}) that must have an initialize (iIc^{st}) logic. The event annotation of a runner (Ev^{R}) must have an initialize (iEv^{R}), process (pEv^{R}) and update (uEv^{R}) logic. The display annotation of a net (D^{n}) must have an initialize (Di^{n}) logic. The event annotation of the net (Ev^{n}) may have an initialize (iEv^{n}) and process (pEv^{n}) logic.

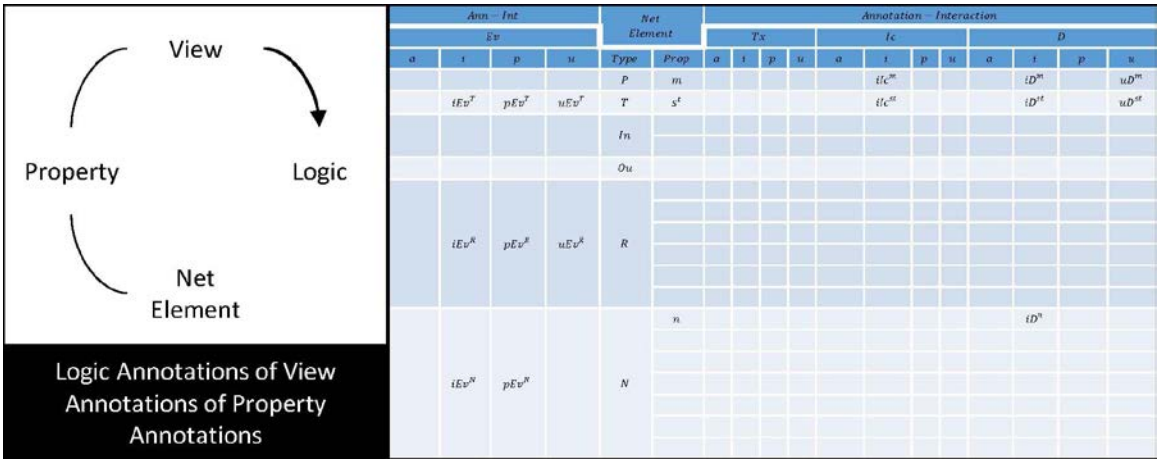

Figure 13 Logic Annotations of View Annotations of Property Annotations

A Guide to Computer Programming

The Net Programming and Integration Workflows (Figure 5) may be broken down into five steps:

1. Draw Petri Net diagram.
2. Create supporting graphics (e.g. tokens for place marks, colours for transition status).
3. Using the Petri Net diagram and the template Form Views as guides:
    a. Create Logic and View Annotations Form
        i. Create net element and annotation headings.
        ii. Generate net element identifiers.
        iii. Create annotation identifiers.
    b. Create Annotation-Interaction Form
        i. Create net element and annotation headings.
        ii. Generate net element identifiers.
        iii. Create annotation identifiers.
4. Using the Petri Net diagram and the filled-in forms as guides:
    a. Create JavaScript program.
5. Integrate PDF documents and JavaScript program.

A vector-based graphics editor may be used to draw a Petri Net diagram. Software capable of editing tables may be used to create annotation forms. And a text editor may be used to create JavaScript programs. To facilitate the creation of annotation forms, the addition of form entries, and the creation of JavaScript programs: a template for logic and view annotations form, a template for annotation-interaction form template, and a naming convention for identifiers may be used as guides.

Logic and View Annotations Form Template. The following is a template for logic and view annotations for net elements, a special form view with generic identifiers. A generic identifier is a prototype naming convention that can be completed by adding element identifiers (such as values from ID, eID, pID, or tID) as subscripts to the prototype. The identifier may be mapped to a JavaScript identifier as follows: $[interaction] < annotation >< property/type >< identifier >$.

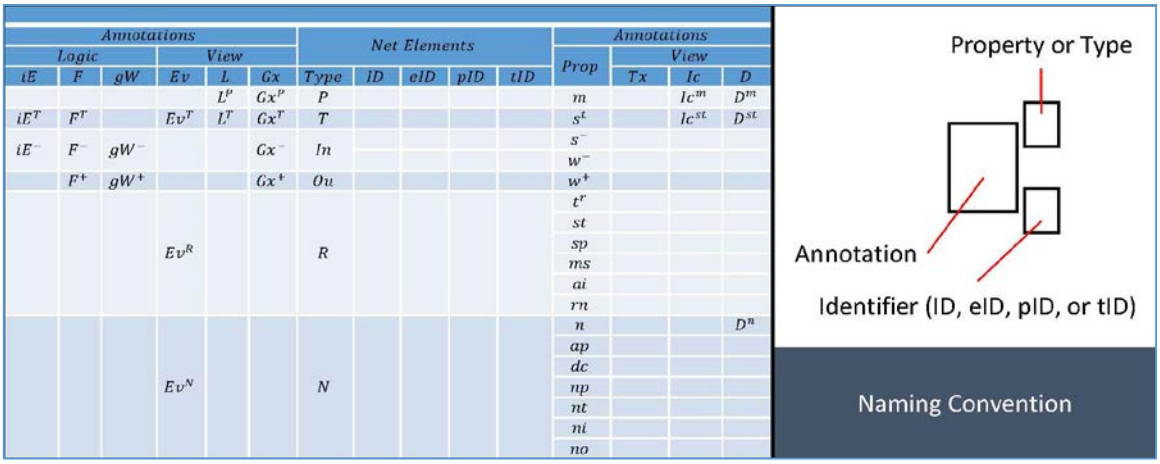

Figure 14 A Template for Net Element Annotations and a Naming Convention

Annotation-Interaction Form Template. Additional logic annotations are needed to handle interactions for view annotations. Figure 15 is a summary of logic annotations for interactions that may be used to create token games and other form-based applications.

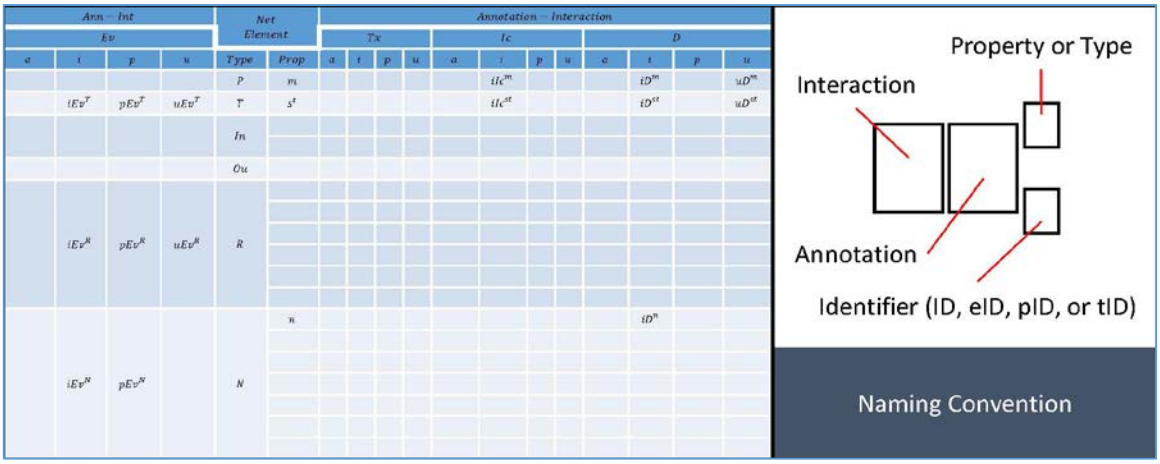

Figure 15 Net Elements View Annotation-Interaction Form

The following rules may also help implement the JavaScript program:

1. A Petri Net model may be implemented as a constructor.
2. There are two ways to implement the property annotations. Each property group (e.g. m, s^t, w^-, s^-, w^+, etc.) may be an array or each property (e.g. m_0, m_1, ..., s_0^t, s_1^t, ..., s_0^-, s_1^-, ..., w_0^-, w_1^-, ...) is a direct reference to the actual property.
3. Properties (including all the properties of Runner (R) and Net (N)) that are not modified by the logic annotations (e.g. np, nt, ni, no, etc.) should be implemented as prototypes. Each group of logic annotation (e.g. iE^T, iE^-, F^T, etc.) may be an array of functions or each logic annotation (e.g. iE_0^T, iE_1^T, iE_2^T, etc.) is a direct reference to a function.
4. Logic annotations should be implemented as function prototypes of the constructor.

Example: A Single Server Queuing System

Consider a single server queuing system based on the description of the Simulation of a Single-Server Queuing System [11]: Customers arrive at a service station and form a queue. There are three servers at the service station. When a server is available, the customer in front of the queue may begin service. It takes a random amount of time for the server to process the service. After the service, the customer leaves the service station and the server is available for the next customer. When the fifth customer begins service, the system stops. Figure 17 is a Place-Transition Net diagram of this queuing system. The related forms and source code follow.

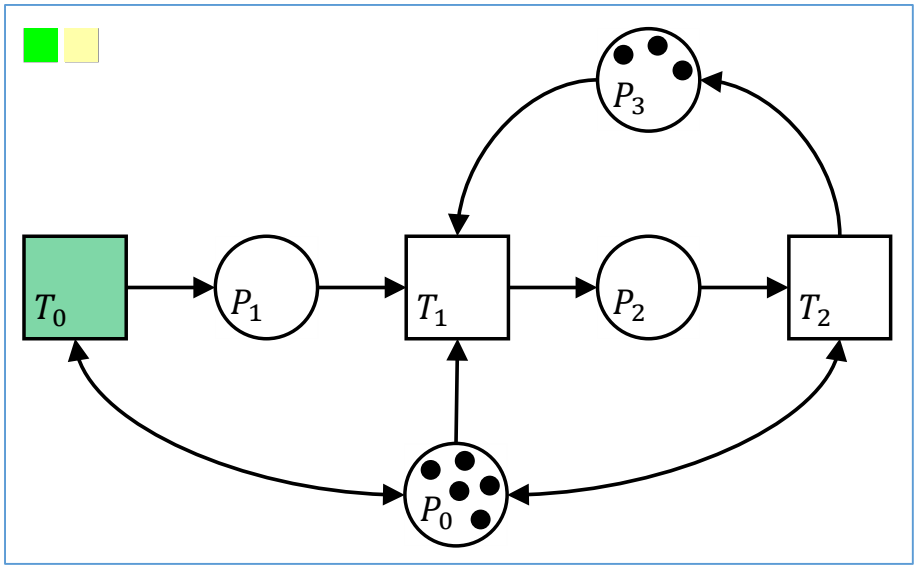

Figure 16 A Place-Transition Net Diagram of a Single Server Queuing System

Single Server Queuing System: Logic and View Annotations														
Annotations						*Net Elements*					*Prop*	*Annotations*		
Logic			*View*									*View*		
iE	*F*	*gW*	*Ev*	*L*	*Gx*	*Type*	*ID*	*eID*	*pID*	*tID*		*Tx*	*Ic*	*D*
				P_0		*P*	0		0		m_0		Ic_0^m	D_0^m
				P_1			1		1		m_1		Ic_1^m	D_1^m
				P_2			2		2		m_2		Ic_2^m	D_2^m
				P_3			3		3		m_3		Ic_3^m	D_3^m
iE_0^T	F_0^T		Ev_0^T	T_0		*T*	4			0	s_0^t		Ic_0^{st}	D_0^{st}
iE_1^T	F_1^T		Ev_1^T	T_1			5			1	s_1^t		Ic_1^{st}	D_1^{st}
iE_2^T	F_2^T		Ev_2^T	T_2			6			2	s_2^t		Ic_2^{st}	D_2^{st}
iE_0^-	F_0^-					*In*	7	0	0	0	w_0^-			
											s_0^-			
iE_1^-	F_1^-						8	1	0	1	w_1^-			
											s_1^-			
iE_2^-	F_2^-						9	2	1	1	w_2^-			
											s_2^-			
iE_3^-	F_3^-						10	3	3	1	w_3^-			
											s_3^-			
iE_4^-	F_4^-						11	4	0	2	w_4^-			
											s_4^-			
iE_5^-	F_5^-						12	5	2	2	w_5^-			
											s_5^-			
	F_0^+					*Ou*	13	0	0	0	w_0^+			
	F_1^+						14	1	1	0	w_1^+			
	F_2^+						15	2	2	1	w_2^+			
	F_3^+						16	3	0	2	w_3^+			
	F_4^+						17	4	3	2	w_4^+			
			Ev_0^R	R_0		*R*	18	0			s^r			
											t^r			
											st			
											sp			

											ms			
											ai			
											rn			
						N	19	0			n			D^n
											ap			
											dc			
											np			
											nt			
											ni			
											no			
											id			

Single Server Queuing System: *Event Annotation − Interaction*								
Annotation − Interaction *Ev*				*Net Elements*				
a	i	p	u	*Type*	*ID*	*eID*	*pID*	*tID*
				P	0		0	
					1		1	
					2		2	
					3		3	
	iEv_0^T	pEv_0^T	uEv_0^T	*T*	4			0
	iEv_1^T	pEv_1^T	uEv_1^T		5			1
	iEv_2^T	pEv_2^T	uEv_2^T		6			2
	iEv_0^R	pEv_0^R	uEv_0^R	*R*	18	0		
	iEv_0^N	pEv_0^N		*N*	19	0		

Single Server Queuing System: *Text, Icon, Display Annotation − Interaction*																	
Net Elements					*Annotation − Interaction*												
					Prop	*Tx*				*Ic*				*D*			
Type	*ID*	*eID*	*pID*	*tID*		a	i	p	u	a	i	p	u	a	i	p	u
	0		0		m_0						iIc_0^m				iD_0^m		uD_0^m
	1		1		m_1						iIc_1^m				iD_1^m		uD_1^m
	2		2		m_2						iIc_2^m				iD_2^m		uD_2^m
	3		3		m_3						iIc_3^m				iD_3^m		uD_3^m
	4			0	s_0^t						iIc_0^{st}				iD_0^{st}		uD_0^{st}
	5			1	s_1^t						iIc_1^{st}				iD_1^{st}		uD_1^{st}
	6			2	s_2^t						iIc_2^{st}				iD_2^{st}		uD_2^{st}
	18	0															
	19	0			n												uD_2^n

Single Server Queuing System: *Sample JavaScript Source Code*		

function SSQ1(parms) {
  var i;
  this.id = parms.id;
  this.m = [];
  for (i=0; i<this.np; i++)
  this.m[i] = 0;
  this.s_t = [];
  for (i=0; i<this.nt; i++)
	prototype.uD_s_t_1 = function() {
	  if (this.s_t[1])
	  this.D_s_t[1].buttonSetIcon(
	this.Ic_s_t[1][1]);
	  else
	  this.D_s_t[1].buttonSetIcon(
	this.Ic_s_t[1][0]);
	  }
		prototype.pEv_T_1 = function() {
		  this.F_T_1();
		  this.iE_T_0();
		  this.iE_T_1();
		  this.iE_T_2();
		  this.uD_m_0();
		  this.uD_m_1();

```
    this.s_t[i] = undefined;
  this.w_in = [];
  this.s_in = [];
  for (i=0; i<this.ni; i++) {
    this.w_in[i] = 1;
    this.s_in[i] = undefined;
  }
  this.w_ou = [];
  for (i=0; i<this.no; i++)
    this.w_ou[i] = 1;
  this.sr = undefined;
  this.tr = [];
  this.etr = undefined;
  this.ms = 1000;
  this.ai = undefined;
  this.D_n = [];
  this.D_m = [];
  this.Ic_m = [];
  this.Ev_T = [];
  this.D_s_t = [];
  this.Ic_s_t = [];
  this.Ev_R = [];
}
with (SSQ1) {
  prototype.n = 0;
  prototype.ap = app;
  prototype.dc = this;
  prototype.np = 4;
  prototype.nt = 3;
  prototype.ni = 6;
  prototype.no = 5;
  prototype.iD_n = function() {
    this.D_n[0] =
this.dc.getField("SSQ1$" + this.id
+ "$D_n$" + 0);
  }
  prototype.iD_m = function() {
    var i;
    for (i=0; i<this.np; i++)
      this.D_m[i] =
this.dc.getField("SSQ1$" + this.id
+ "$D_m$" + i);
  }
  prototype.iIc_m = function() {
    var i, j;
    for (i=0; i<this.np; i++) {
      this.Ic_m[i] = [];
      for (j=0; j<7; j++)
        this.Ic_m[i][j] =
this.dc.getField("Token$" +
j).buttonGetIcon();
    }
  }
  prototype.iD_s_t = function() {
    var i;
    for (i=0; i<this.nt; i++)
      this.D_s_t[i] =
this.dc.getField("SSQ1$" + this.id
+ "$D_s_t$" + i);
  }
  prototype.iIc_s_t = function() {
    var i;
    for (i=0; i<this.nt; i++) {
  prototype.uD_s_t_2 = function() {
    if (this.s_t[2])
      this.D_s_t[2].buttonSetIcon(
this.Ic_s_t[2][1] );
    else
      this.D_s_t[2].buttonSetIcon(
this.Ic_s_t[2][0] );
  }
  prototype.uEv_T_0 = function() {
    if (this.s_t[0])
      this.Ev_T[0].display =
display.visible;
    else
      this.Ev_T[0].display =
display.hidden;
  }
  prototype.uEv_T_1 = function() {
    if (this.s_t[1])
      this.Ev_T[1].display =
display.visible;
    else
      this.Ev_T[1].display =
display.hidden;
  }
  prototype.uEv_T_2 = function() {
    if (this.s_t[2])
      this.Ev_T[2].display =
display.visible;
    else
      this.Ev_T[2].display =
display.hidden;
  }
  prototype.iE_In_0 = function() {
    if (this.m[0] < 1)
      this.s_in[0] = false;
    else
      this.s_in[0] = true;
  }
  prototype.iE_In_1 = function() {
    if (this.m[0] < 1)
      this.s_in[1] = false;
    else
      this.s_in[1] = true;
  }
  prototype.iE_In_2 = function() {
    if (this.m[1] < 1)
      this.s_in[2] = false;
    else
      this.s_in[2] = true;
  }
  prototype.iE_In_3 = function() {
    if (this.m[3] < 1)
      this.s_in[3] = false;
    else
      this.s_in[3] = true;
  }
  prototype.iE_In_4 = function() {
    if (this.m[0] < 1)
      this.s_in[4] = false;
    else
      this.s_in[4] = true;
  }
  prototype.iE_In_5 = function() {
    this.uD_m_2();
    this.uD_m_3();
    this.uD_s_t_0();
    this.uD_s_t_1();
    this.uD_s_t_2();
    this.uEv_T_0();
    this.uEv_T_1();
    this.uEv_T_2();
  }
  prototype.pEv_T_2 = function() {
    this.F_T_2();
    this.iE_T_1();
    this.iE_T_2();
    this.uD_m_2();
    this.uD_m_3();
    this.uD_s_t_1();
    this.uD_s_t_2();
    this.uEv_T_1();
    this.uEv_T_2();
  }
  prototype.pEv_N_0 = function() {
    this.iD_n();
    this.iD_m();
    this.iIc_m();
    this.iD_s_t();
    this.iIc_s_t();
    this.iEv_T();
    this.iEv_R();
    var i;
    this.m[0] = 5;
    this.m[1] = 0;
    this.m[2] = 0;
    this.m[3] = 2;
    for (i=0; i<this.np; i++)
      eval("this.uD_m_" + i + "();");
    for (i=0; i<this.nt; i++) {
      eval("this.iE_T_" + i + "();");
      eval("this.uD_s_t_" + i + "();");
      eval("this.uEv_T_" + i + "();");
    }
    if (this.ai)
      this.sp();
    this.Ev_R[0].fillColor =
color.green;
  }
  prototype.rn = function(){
    var i;
    this.etr = [];
    for (i=0; i<this.nt; i++) {
      if (this.s_t[i])
        this.etr[this.etr.length] = i;
    }
    if (this.etr.length>0) {
      i = Math.round((this.etr.length-
1)*Math.random());
      switch(this.etr[i]) {
        case 0:
          this.pEv_T_0();
          break;
        case 1:
          this.pEv_T_1();
          break;
        case 2:
```

```
      this.Ic_s_t[i] = [];
      this.Ic_s_t[i][0] =
this.dc.getField("Colour$NotEnable
d").buttonGetIcon();
      this.Ic_s_t[i][1] =
this.dc.getField("Colour$1").button
GetIcon();
    }
  }
  prototype.iEv_T = function() {
    var i;
    for (i=0; i<this.nt; i++)
      this.Ev_T[i] =
this.dc.getField("SSQ1$" + this.id
+ "$Ev_T$" + i);
  }
  prototype.iEv_R = function() {
    this.Ev_R[0] =
this.dc.getField("SSQ1$" + this.id
+ "$Ev_R$0");
  }
  prototype.uD_m_0 = function() {
    if (this.m[0] <
this.Ic_m[0].length)
      this.D_m[0].buttonSetIcon(
        this.Ic_m[0][this.m[0]]
      );
    else
      this.D_m[0].buttonSetIcon(

this.Ic_m[0][this.Ic_m[0].length-1]
      );
  }
  prototype.uD_m_1 = function() {
    if (this.m[1] <
this.Ic_m[1].length)
      this.D_m[1].buttonSetIcon(
        this.Ic_m[1][this.m[1]]
      );
    else
      this.D_m[1].buttonSetIcon(

this.Ic_m[1][this.Ic_m[1].length-1]
      );
  }
  prototype.uD_m_2 = function() {
    if (this.m[2] <
this.Ic_m[2].length)
      this.D_m[2].buttonSetIcon(
        this.Ic_m[2][this.m[2]]
      );
    else
      this.D_m[2].buttonSetIcon(

this.Ic_m[2][this.Ic_m[2].length-1]
      );
  }
  prototype.uD_m_3 = function() {
    if (this.m[3] <
this.Ic_m[3].length)
      this.D_m[3].buttonSetIcon(
        this.Ic_m[3][this.m[3]]
      );
    if (this.m[2] < 1)
      this.s_in[5] = false;
    else
      this.s_in[5] = true;
  }
  prototype.iE_T_0 = function() {
    this.iE_In_0();
    this.s_t[0] = this.s_in[0];
  }
  prototype.iE_T_1 = function() {
    this.iE_In_1();
    this.iE_In_2();
    this.iE_In_3();
    this.s_t[1] = this.s_in[1] &&
this.s_in[2] && this.s_in[3];
  }
  prototype.iE_T_2 = function() {
    this.iE_In_4();
    this.iE_In_5();
    this.s_t[2] = this.s_in[4] &&
this.s_in[5];
  }
  prototype.F_In_0 = function() {
    this.m[0] -= 1;
  }
  prototype.F_In_1 = function() {
    this.m[0] -= 1;
  }
  prototype.F_In_2 = function() {
    this.m[1] -= 1;
  }
  prototype.F_In_3 = function() {
    this.m[3] -= 1;
  }
  prototype.F_In_4 = function() {
    this.m[0] -= 1;
  }
  prototype.F_In_5 = function() {
    this.m[2] -= 1;
  }
  prototype.F_Ou_0 = function() {
    this.m[0] += 1;
  }
  prototype.F_Ou_1 = function() {
    this.m[1] += 1;
  }
  prototype.F_Ou_2 = function() {
    this.m[2] += 1;
  }
  prototype.F_Ou_3 = function() {
    this.m[0] += 1;
  }
  prototype.F_Ou_4 = function() {
    this.m[3] += 1;
  }
  prototype.F_T_0 = function() {
    this.F_Ou_1();
  }
  prototype.F_T_1 = function() {
    this.F_In_1();
    this.F_In_2();
    this.F_In_3();
    this.F_Ou_2();
      this.pEv_T_2();
      break;
    default:
      break;
    }
  } else
    this.sp();
  }
  prototype.uEv_R_0 = function() {
    if (this.ai)
      this.Ev_R[0].fillColor =
color.red;
    else
      this.Ev_R[0].fillColor =
color.green;
  }
  prototype.st = function() {
    this.sr = true;
    this.ai =
this.ap.setInterval("agent[" +
this.id + "].rn();", this.ms);
    this.uEv_R_0();
  }
  prototype.sp = function() {
    this.ap.clearInterval(this.ai);
    this.ai = undefined;
    this.sr = undefined;
    this.etr = undefined;
    this.uEv_R_0();
  }
  prototype.pEv_R_0 = function() {
    if (this.ai)
      this.sp();
    else
      this.st();
  }
}
var agent= [];
(function() {
  var id = 1401414769;
  agent[id] = new SSQ1({id: id});
  with (agent[id]) {
    pEv_N_0();
  }
})();
```

```
    else
      this.D_m[3].buttonSetIcon(

this.Ic_m[3][this.Ic_m[3].length-1]
      );
  }
  prototype.uD_s_t_0 = function() {
    if (this.s_t[0])
      this.D_s_t[0].buttonSetIcon(
this.Ic_s_t[0][1] );
    else
      this.D_s_t[0].buttonSetIcon(
this.Ic_s_t[0][0] );
  }
```

```
}
 prototype.F_T_2 = function() {
   this.F_In_5();
   this.F_Ou_4();
 }
 prototype.pEv_T_0 = function() {
   this.F_T_0();
   this.iE_T_1();
   this.uD_m_1();
   this.uD_s_t_1();
   this.uEv_T_1();
 }
```

Conclusion

With a Petri Net diagram and the annotation system, it is possible to systematically create JavaScript programs. A Petri Net diagram is a useful guide for systematically establishing the input-output relations of places and transitions. The annotation system is a useful companion to a Petri Net diagram for systematically building the properties and logic computations of an application. The annotations may be added to the net elements or net element properties. The addition may be performed in a database, a form or objects of a graphics editor. To facilitate the annotation process, several classes of annotations, a form view of the annotations, and a form view of the annotation-interactions are included in the annotation system. The computer programs are JavaScript programs that run in PDF documents. They interact with AcroForm objects (app, Document and Field objects) using the Acrobat/JavaScript API.

References

1. Adobe Systems Incorporated. (2012). Adobe Acrobat XI/9 [software]. San Jose, California: Adobe Systems Incorporated.

2. Adobe Systems Incorporated. (2007). Adobe Acrobat SDK 8.1 JavaScript for Acrobat API Reference for Microsoft Windows and Mac OS. Edition 2.0, April 2007. San Jose, California: Adobe Systems Incorporated. Retrieved Aug. 3, 2010 from http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf.

3. Adobe Systems Incorporated. (2006). Adobe Acrobat SDK 8.0 Developing Acrobat Applications Using JavaScript for Microsoft Windows and Mac OS. Edition 1.0, November 2006. San Jose, California: Adobe Systems Incorporated. Retrieved Aug. 6, 2010 from http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_developer_guide.pdf.

4. David, R. and H. Alla. (1992). Petri Nets and Grafcet: Tools for Modeling Discrete-Event Systems. Upper Saddle, NJ: Prentice Hall.

5. Denaro, G. and Pezzè, M. (2004). Petri Nets and Software Engineering [electronic version] . Desel, J., Reisig, W., and Rozenberg, G. (Eds.): ACPN 2003, LNCS 3098, p. 439 – 466, 2004. Retrieved on Feb. 2, 2014 from http://www.inf.uni-konstanz.de/soft/teaching/ws13/seminar/9.pdf.

6. Flanagan, D. (2006). JavaScript: The Definitive Guide, Fifth Edition. Sebastopol, CA: O'Reilly Media Inc.

7. Genrich, H. J. and Lautenbach, K. (1981). System Modelling with High-Level Petri Nets. Theoretical Computer Science, vol. 13, North-Holland Publishing Company, pp. 109 – 136. Retrieved on May 1, 2014 from http://www.sciencedirect.com/science/article/pii/0304397581901134.

8. Jensen, K. and Kristensen, L. M. (2009). Net Structure and Inscriptions (Chapter 4.2) and Enabling and Occurrence of Steps (Chapter 4.3) in Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Berlin, Heidelberg: Springer-Verlag.

9. Menzies, T. (2002). Evaluation issues for visual programming languages. In S. K. Chang (Ed). Handbook of Software Engineering & Knowledge Engineering, Vol. 2 Emerging Technologies. World Scientific Publishing co. Pte. Ltd., pp. 93 – 101.

10. Holt, A. W. and Commoner, F. (1969). Events and Conditions: An Approach to the Description and Analysis of Dynamic Systems [electronic version]. Third Semi-Annual Technical Report, Part II (Covering Task Area II) (22 June 1969 – 21 December 1969) For the Project "Research in Machine-Independent Software Programming", Wakefield, MA: Massachusetts Computer Associates, a Division of Applied Data Research Inc.

11. Law, A. M. (2007). Simulation of a Single-Server Queuing System (Chapter 1.4) in Simulation Modeling and Analysis, 4th ed. New York: McGraw-Hill.

12. Lin, Z.; Wen, F.; and Chung, C. Y. (2006). A Survey on the Applications of Petri Net Theory in Power Systems. Power Engineering Society General Meeting, 2006. IEEE. DOI: 10.1109/PES.2006.1709115.

13. Microsoft (2013a). PowerPoint 2013/2007 [software]. Redmond, WA: Microsoft Corporation.

14. Microsoft (2013b) Word 2013/2007 [software]. Redmond, WA: Microsoft Corporation.

15. Murata, T. (1989). "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, vol. 77 no. 4, April 1989, pp. 541 – 580.

16. Noe, J. D. and Nutt, G. J. (1973). "Macro E-Nets for Representation of Parallel Systems", IEEE Transactions on Computers, vol. C-22, No. 8, Aug. 1973, pp. 718 – 727.

17. Peterson, J. L. (1977). "Petri Nets", ACM Computing Surveys, vol. 9, no. 3, pp. 223 – 252, Sept. 1977.

18. Petri, C.A. (1980). Introduction to General Net Theory. Lecture Notes in Computer Science Vol. 84: Net Theory and Applications, Proc. Of the Advanced Course on General Net Theory of Processing and Systems, Hamburg, 1979 / Brauer, W. (ed.). Berlin-Heidelberg-New York: Spring-Verlag, pp. 1-19 (1980).

19. Petri, C. A. (1973). Concepts of Net Theory. In Mathematical Foundations of Computer Science: Proc. of Symposium and Summer School, High Tatras, Sep. 3 – 8, 1973, pages 137 – 146. Math. Inst. of the Slovak Acad. of Sciences, 1973.

20. Petri, C. A. (1966). Communication with Automota [trans. C.F. Greene, Jr.]. Supplement I to Technical Report RADC-TR-65-377 (Volume I). Griffiss Air Force Base, NY: Rome Air Development Center, Research and Technology Division, Air Force Systems Command, Griffiss Air Force Base. Retrieved Aug. 31, 2011 from http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri/doc/Petri-diss-engl.pdf.

21. Reisig, W. (1991). Petri nets and Algebraic Specifications. Theoretical Computer Science, vol. 80, iss. 1, March 21, 1991, pp. 1 – 34. Retrieved on May 20, 2014 from http://www.sciencedirect.com/science/article/pii/030439759190203E.